浙江大学

本科实验报告

课程名称：操作系统

姓　　名：

学　　院：计算机科学与技术学院

系：计算机科学与技术系

专　　业：计算机科学与技术

学　　号：

指导教师：李环

2024 年 11 月 27 日

浙江大学操作系统实验报告

实验名称：RV64 用户态程序

电子邮件地址：____手机：_____

实验地点：曹西 503 实验日期：2024 年 11 月 27 日

一、实验目的和要求

创建用户态进程，并完成内核态与用户态的转换

正确设置用户进程的用户态栈和内核态栈，并在异常处理时正确切换

补充异常处理逻辑，完成指定的系统调用（SYS_WRITE, SYS_GETPID）功能

实现用户态 ELF 程序的解析和加载

二、实验过程

（一）准备工程

需要修改 vmlinux.lds，将用户态程序 uapp 加载至 .data 段：

```
1.      .data : ALIGN(0x1000) {
2.          _sdata = .;
3.
4.          *(.sdata .sdata*)
5.          *(.data .data.*)
6.
7.          _edata = .;
8.
9.          . = ALIGN(0x1000);
10.         _sramdisk = .;
11.         *(.uapp .uapp*)
12.         _eramdisk = .;
13.         . = ALIGN(0x1000);
14.     } >ramv AT>ram
```

需要修改 defs.h，在 defs.h 添加如下内容：

```
#define USER_START (0x0000000000000000) // user space start virtual address
#define USER_END (0x0000004000000000) // user space end virtual address
```

从本仓库同步以下文件和文件夹，并按照下面的位置来放置这些新文件：

```
.
├── arch
│   └── riscv
│       ├── Makefile     # 添加了链接 uapp 的部分
│       ├── include
│       │   └── mm.h     # 修改为了 buddy system
│       └── kernel
│           └── mm.c     # 修改为了 buddy system
├── include
│   └── elf.h            # 来自 musl libc
└── user
    ├── Makefile
    ├── getpid.c         # 用户态程序
    ├── link.lds
    ├── printf.c         # 类似 printk
    ├── start.S          # 用户态程序入口
    ├── stddef.h
    ├── stdio.h
    ├── syscall.h
    └── uapp.S           # 提取二进制内容
```

修改根目录下的 Makefile，将 user 文件夹下的内容纳入工程管理

```
all: clean
 ${MAKE} -C lib all
 ${MAKE} -C init all
 ${MAKE} -C user all
 ${MAKE} -C arch/riscv all
 @echo -e '\n'Build Finished OK
......
clean:
 ${MAKE} -C lib clean
 ${MAKE} -C init clean
 ${MAKE} -C user clean
 ${MAKE} -C arch/riscv clean
 $(shell test -f vmlinux && rm vmlinux)
```

```
    $(shell test -f System.map && rm System.map)
    @echo -e '\n'Clean Finished
```

（二）创建用户态进程

1. 结构体更新

本次实验只需要创建 4 个用户态进程，修改 proc.h 中的 NR_TASKS 即可

由于创建用户态进程要对 sepc , sstatus , sscratch 做设置，我们需要将其加入 thread_struct 中

由于多个用户态进程需要保证相对隔离，因此不可以共用页表，我们需要为每个用户态进程都创建一个页表并记录在 task_struct 中，可以参考的修改如下：

```
struct thread_struct {
    uint64_t ra;
    uint64_t sp;
    uint64_t s[12];
    uint64_t sepc, sstatus, sscratch;
};

struct task_struct {
    uint64_t state;
    uint64_t counter;
    uint64_t priority;
    uint64_t pid;

    struct thread_struct thread;
    uint64_t *pgd;  // 用户态页表
};
```

2. 修改 task_init()

对于每个进程，初始化我们刚刚在 thread_struct 中添加的三个变量，具体而言：

将 sepc 设置为 USER_START

配置 sstatus 中的 SPP（使得 sret 返回至 U-Mode）、SPIE（sret 之后开启中断）、SUM（S-Mode 可以访问 User 页面）

将 sscratch 设置为 U-Mode 的 sp，其值为 USER_END（将用户态栈放置在 user space 的最后一个页面）

对于每个进程，创建属于它自己的页表：

为了避免 U-Mode 和 S-Mode 切换的时候切换页表，我们将内核页表 swapper_pg_dir 复制到每个进程的页表中

将 uapp 所在的页面映射到每个进程的页表中

对于每个进程，分配一块新的内存地址，将 uapp 二进制文件拷贝过去

设置用户态栈，对每个用户态进程，其拥有两个栈：

用户态栈：我们可以申请一个空的页面来作为用户态栈，并映射到进程的页表中

内核态栈：在 lab3 中已经设置好了，就是 thread.sp

```
        for (int i = 1; i < NR_TASKS; i++){
            task[i] = (struct task_struct*) kalloc();
            task[i]->state = TASK_RUNNING;
            task[i]->counter = 0;
            task[i]->priority = rand() % (PRIORITY_MAX - PRIORITY_MIN + 1) + PRIORITY_MIN;
            task[i]->pid = i;
            task[i]->thread.ra = (uint64_t) &__dummy;
            task[i]->thread.sp = (uint64_t) task[i] + PGSIZE;

            task[i]->thread.sepc = USER_START;
            task[i]->thread.sstatus = csr_read(sstatus);
            task[i]->thread.sstatus &= ~(1 << 8);
            task[i]->thread.sstatus |= (1 << 5);
            task[i]->thread.sstatus |= (1 << 18);
            task[i]->thread.sscratch = USER_END;

            task[i]->pgd = (uint64_t)alloc_page();
            for (int j = 0; j < 512; j++){
                task[i]->pgd[j] = swapper_pg_dir[j];
            }

            load_program(task[i]);

            uint64_t U_stack_top = kalloc();

            uint64_t size = PGROUNDUP((uint64_t)_eramdisk - (uint64_t)_sramdisk) / PGSIZE;
            uint64_t copy_addr = alloc_pages(size);
            for (int j = 0; j < _eramdisk - _sramdisk; ++j)
                ((char *)copy_addr)[j] = _sramdisk[j];

            create_mapping(task[i]->pgd, USER_START, (uint64_t)copy_addr - PA2VA_OFFSET, size * PGSIZE, 31); // 映射用户段   U|X|W|R|V
            create_mapping(task[i]->pgd, USER_END - PGSIZE, U_stack_top - PA2VA_OFFSET, PGSIZE, 23); // 映射用户栈 U|-|W|R|V
        }
```

（三）修改__switch_to

在前面新增了 sepc、sstatus、sscratch 之后，需要将这些变量在切换进程时保存在栈上，因此需要更新 __switch_to 中的逻辑，同时需要增加切换页表的逻辑。在切换了页表之后，需要通过 sfence.vma 来刷新 TLB 和 ICache。

```
1.    __switch_to:
2.        addi t0, a0, 32 # save state to prev process
3.        sd ra, 0*8(t0)
4.        sd sp, 1*8(t0)
5.        sd s0, 2*8(t0)
6.        sd s1, 3*8(t0)
7.        sd s2, 4*8(t0)
8.        sd s3, 5*8(t0)
9.        sd s4, 6*8(t0)
10.       sd s5, 7*8(t0)
11.       sd s6, 8*8(t0)
12.       sd s7, 9*8(t0)
13.       sd s8, 10*8(t0)
14.       sd s9, 11*8(t0)
15.       sd s10, 12*8(t0)
16.       sd s11, 13*8(t0)
17.       csrr t1, sepc
18.       sd t1, 14*8(t0)
19.       csrr t1, sstatus
20.       sd t1, 15*8(t0)
21.       csrr t1, sscratch
22.       sd t1, 16*8(t0)
23.
24.       addi t0, a1, 32 # restore state from next process
25.       ld ra, 0(t0)
26.       ld sp, 8(t0)
27.       ld s0, 2*8(t0)
28.       ld s1, 3*8(t0)
29.       ld s2, 4*8(t0)
30.       ld s3, 5*8(t0)
31.       ld s4, 6*8(t0)
32.       ld s5, 7*8(t0)
33.       ld s6, 8*8(t0)
34.       ld s7, 9*8(t0)
35.       ld s8, 10*8(t0)
36.       ld s9, 11*8(t0)
37.       ld s10, 12*8(t0)
38.       ld s11, 13*8(t0)
```

```
    ld t1, 14*8(t0)
    csrw sepc, t1
    ld t1, 15*8(t0)
    csrw sstatus, t1
    ld t1, 16*8(t0)
    csrw sscratch, t1

    addi t0, zero, 1
    slli t0, t0, 63
    ld t1, 168(a1)
    li t2, PA2VA_OFFSET
    sub t1, t1, t2
    srli t1, t1, 12
    or t0, t0, t1
    csrw satp, t0

    sfence.vma zero, zero

    fence.i

    ret
```

（四）更新中断处理逻辑

与 ARM 架构不同的是，RISC-V 中只有一个栈指针寄存器 sp，因此需要我们来完成用户栈与内核栈的切换。

由于我们的用户态进程运行在 U-Mode 下，使用的运行栈也是用户栈，因此当触发异常时，我们首先要对栈进行切换（从用户栈切换到内核栈）。同理，当我们完成了异常处理，从 S-Mode 返回至 U-Mode 时，也需要进行栈切换（从内核栈切换到用户栈）。

1. 修改__dummy

在我们初始化线程时，thread_struct.sp 保存了内核态栈 sp，thread_struct.sscratch 保存了用户态栈 sp，因此在 __dummy 进入用户态模式的时候，我们需要切换这两个栈，只需要交换对应的寄存器的值即可。

```
__dummy:
    csrr t0, sscratch
    csrw sscratch, sp
    mv sp, t0
    sret
```

2. 修改_traps

同理，在 _traps 的首尾我们都需要做类似的操作，进入 trap 的时候需要切换到内核栈，处理完成后需要再切换回来。

注意如果是内核线程（没有用户栈）触发了异常，则不需要进行切换。（内核线程的 sp 永远指向的内核栈，且 sscratch 为 0）

```
1.    _traps:
2.        csrr t0, sscratch
3.        beqz t0, _S_mode
4.        csrw sscratch, sp
5.        mv sp, t0
6.
7.    _S_mode:
8.        addi sp,sp,-34*8
9.        sd x0,0*8(sp)
10.       sd x1,1*8(sp)
11.       sd x2,2*8(sp)
12.       sd x3,3*8(sp)
13.       sd x4,4*8(sp)
14.       sd x5,5*8(sp)
15.       sd x6,6*8(sp)
16.       sd x7,7*8(sp)
17.       sd x8,8*8(sp)
18.       sd x9,9*8(sp)
19.       sd x10,10*8(sp)
20.       sd x11,11*8(sp)
21.       sd x12,12*8(sp)
22.       sd x13,13*8(sp)
23.       sd x14,14*8(sp)
24.       sd x15,15*8(sp)
25.       sd x16,16*8(sp)
26.       sd x17,17*8(sp)
27.       sd x18,18*8(sp)
28.       sd x19,19*8(sp)
29.       sd x20,20*8(sp)
30.       sd x21,21*8(sp)
31.       sd x22,22*8(sp)
32.       sd x23,23*8(sp)
33.       sd x24,24*8(sp)
34.       sd x25,25*8(sp)
35.       sd x26,26*8(sp)
36.       sd x27,27*8(sp)
37.       sd x28,28*8(sp)
```

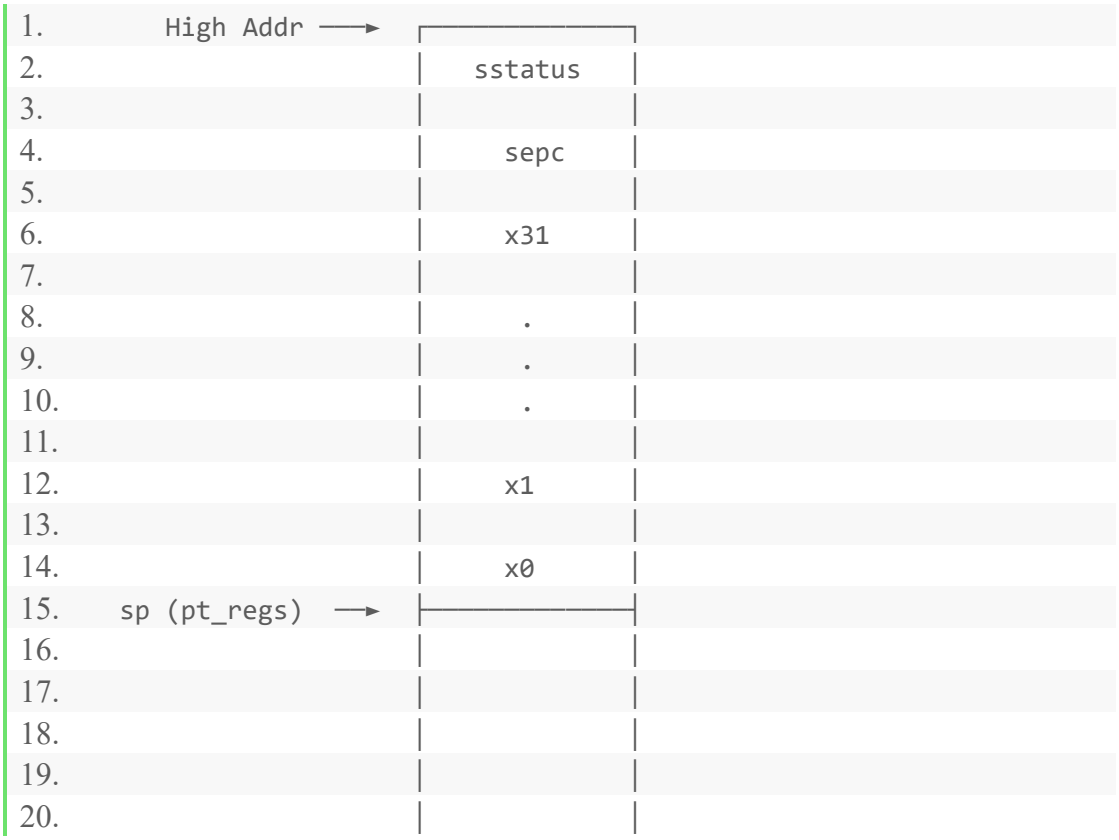
```
38.       sd x29,29*8(sp)
39.       sd x30,30*8(sp)
40.       sd x31,31*8(sp)
41.       csrr t0, sepc
42.       sd t0,32*8(sp)
43.       csrr t0, sstatus
44.       sd t0, 33*8(sp)#save registers and sepc
45.
46.       csrr a0,scause
47.       csrr a1,sepc
48.       mv a2, sp
49.       jal trap_handler #trap handler
50.
51.       ld x0,0*8(sp)
52.       ld x1,1*8(sp)
53.       #ld x2,2*8(sp)
54.       ld x3,3*8(sp)
55.       ld x4,4*8(sp)
56.       ld x5,5*8(sp)
57.       ld x6,6*8(sp)
58.       ld x7,7*8(sp)
59.       ld x8,8*8(sp)
60.       ld x9,9*8(sp)
61.       ld x10,10*8(sp)
62.       ld x11,11*8(sp)
63.       ld x12,12*8(sp)
64.       ld x13,13*8(sp)
65.       ld x14,14*8(sp)
66.       ld x15,15*8(sp)
67.       ld x16,16*8(sp)
68.       ld x17,17*8(sp)
69.       ld x18,18*8(sp)
70.       ld x19,19*8(sp)
71.       ld x20,20*8(sp)
72.       ld x21,21*8(sp)
73.       ld x22,22*8(sp)
74.       ld x23,23*8(sp)
75.       ld x24,24*8(sp)
76.       ld x25,25*8(sp)
77.       ld x26,26*8(sp)
78.       ld x27,27*8(sp)
79.       ld x28,28*8(sp)
80.       ld x29,29*8(sp)
81.       ld x30,30*8(sp)
```

```
82.       ld x31,31*8(sp)
83.       ld t0,32*8(sp)
84.       csrw sepc,t0
85.       ld t0, 33*8(sp)
86.       csrw sstatus, t0
87.       ld x2,2*8(sp)
88.       addi x2,x2,34*8 #restore the value of the registers and sepc
89.
90.       csrr t0, sscratch
91.       beqz t0, __return
92.       csrw sscratch, sp
93.       mv sp, t0
```

3. 修改 trap_handler

uapp 使用 ecall 会产生 Environment Call from U-mode，而且处理系统调用的时候还需要寄存器的值，因此我们需要在 trap_handler() 里面进行捕获。

在 _traps 中我们将寄存器的内容连续的保存在内核栈上，因此我们可以将这一段看做一个叫做 pt_regs 的结构体。我们可以从这个结构体中取到相应的寄存器的值（比如 syscall 中我们需要从 a0 ~ a7 寄存器中取到参数）。这个结构体中的值也可以按需添加，同时需要在 _traps 中存入对应的寄存器值以供使用，示例如下图：



```
                        |              |
                        |              |
                        |              |
                        |              |
        Low  Addr ──►   └──────────────┘
```

修改 trap_handler() 如下：

```
  void trap_handler(uint64_t scause, uint64_t sepc, struct pt_regs *regs) {
      // 通过 `scause` 判断 trap 类型
      // 如果是 interrupt 判断是否是 timer interrupt
      if (scause >> 63 && scause % 8 == 5){
          // 如果是 timer interrupt 则打印输出相关信息，并通过 `clock_set_next_event()` 设置下一次时钟中断
          //printk("[S] Supervisor Mode Timer Interrupt\n");
          // `clock_set_next_event()` 见 4.3.4 节
          clock_set_next_event();
          do_timer();
      } else if (scause == 8) {
          uint64_t ret;
          if (regs->x[17] == SYS_WRITE) {
              ret = sys_write((unsigned int)(regs->x[10]), (const char*)(regs->x[11]), (size_t)(regs->x[12]));
              regs->sepc += 4;
          }
          else if (regs->x[17] == SYS_GETPID) {
              ret = sys_getpid();
              regs->sepc += 4;
          }
          regs->x[10] = ret;
      }
      // 其他 interrupt / exception 可以直接忽略，推荐打印出来供以后调试
  }
```

（五）添加系统调用

本次实验要求的系统调用函数原型以及具体功能如下：

64 号系统调用 sys_write(unsigned int fd, const char* buf, size_t count) 该调用将用户态传递的字符串打印到屏幕上，此处 fd 为标准输出即 1，buf 为用户需要打印的起始地址，count 为字符串长度，返回打印的字符数；

172 号系统调用 sys_getpid() 该调用从 current 中获取当前的 pid 放入 a0 中返回，无参数

增加 syscall.c , syscall.h 文件，并在其中实现 getpid() 以及 write() 逻辑

系统调用的返回参数放置在 a0 中（不可以直接修改寄存器，应该修改 regs 中保存的内容）

针对系统调用这一类异常，我们需要手动完成 sepc + 4

```
#define SYS_WRITE   64
#define SYS_GETPID  172

#include "proc.h"
#include "stddef.h"
#include "stdint.h"

extern struct task_struct* current;

uint64_t sys_write(unsigned int fd, const char* buf, size_t count);
uint64_t sys_getpid();
```

```
#include "syscall.h"
#include "printk.h"

uint64_t sys_write(unsigned int fd, const char* buf, size_t count)
{
    uint64_t length = 0;
    if (fd == 1) {  //standard output
        for (size_t i = 0; i < count; ++i) {
            length += (uint64_t)printk("%c", buf[i]);
        }
    }

    return length;
}

uint64_t sys_getpid()
{
    return current->pid;
}
```

（六）调整时钟中断

接下来我们需要修改 head.S 以及 start_kernel：

在之前的 lab 中，在 OS boot 之后，我们需要等待一个时间片，才会进行调度，我们现在

更改为 OS boot 完成之后立即调度 uapp 运行

即在 start_kernel() 中，test() 之前调用 schedule()

将 head.S 中设置 sstatus.SIE 的逻辑注释掉，确保 schedule 过程不受中断影响

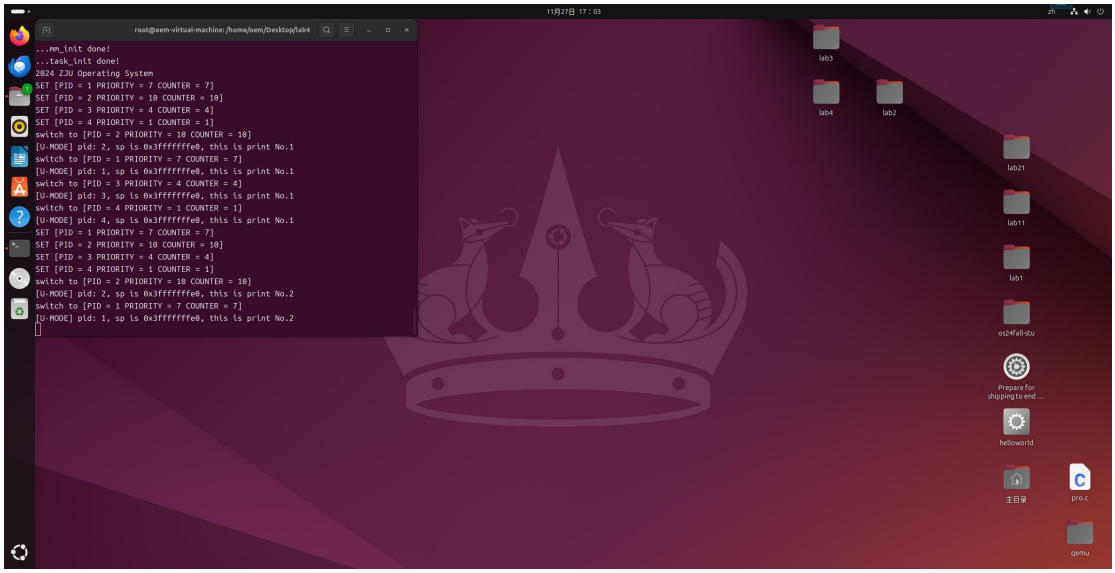
```
int start_kernel() {
    printk("2024");
    printk(" ZJU Operating System\n");

    schedule();
    test();
    return 0;
}
```

```
_start:
    la sp, boot_stack_top #store the address of stack top
    li t0, PA2VA_OFFSET
    sub sp, sp, t0

    call setup_vm
    call relocate
    call mm_init
    call setup_vm_final
    call task_init

    la t0,_traps
    csrw stvec,t0 #stvec = _traps

    csrr t0,sie
    ori t0,t0,0x20
    csrw sie,t0 #sie[STIE] = sie[5] = 1

    andi a1,x0,0
    andi a2,x0,0
    andi a3,x0,0
    andi a4,x0,0
    andi a5,x0,0
    andi a6,x0,0
    andi a7,x0,0
    li t0,10000000
    rdtime a0
    add a0,a0,t0
    ecall #set first time interrupt

```

```
31.       #csrr t0,sstatus
32.       #ori  t0,t0,0x2
33.       #csrw sstatus,t0 #sstatus[SIE] = sstatus[1] = 1
34.
35.       jal start_kernel #jump to start_kernel
```

（七）测试纯二进制文件

测试结果如下：



根据图中显示，测试结果正常。

（八）添加 ELF 解析与加载

首先我们需要将 uapp.S 中的 payload 给换成我们的 ELF 文件：

```
1.    .section .uapp
2.
3.    .incbin "uapp"
```

这时候从 _sramdisk 开始的数据就变成了名为 uapp 的 ELF 文件，也就是说 _sramdisk 处 32-bit 的数据不再是我们需要执行第一条指令了，而是 ELF Header 的开始。

这时候就需要对 task_init 中的初始化步骤进行修改。 ELF 相关的结构体定义如下：

```
1.    typedef struct {
2.      unsigned char e_ident[EI_NIDENT]; // magic number，判断是否是 Ehdr，固定为 7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
```

```
    Elf64_Half    e_type;
    Elf64_Half    e_machine;
    Elf64_Word    e_version;
    Elf64_Addr    e_entry;        // 程序的第一条指令被存储的用户态虚拟地址
    Elf64_Off e_phoff;            // ELF 文件包含的 Segment 数组（Phdr）相对于 Ehdr 的偏移量
    Elf64_Off e_shoff;
    Elf64_Word    e_flags;
    Elf64_Half    e_ehsize;
    Elf64_Half    e_phentsize;
    Elf64_Half    e_phnum;        // ELF 文件包含的 Segment 的数量
    Elf64_Half    e_shentsize;
    Elf64_Half    e_shnum;
    Elf64_Half    e_shstrndx;
} Elf64_Ehdr;

typedef struct {
    Elf64_Word    p_type;     // Segment 的类型
    Elf64_Word    p_flags;    // Segment 的权限（包括了读、写和执行）
    Elf64_Off p_offset;       // Segment 在文件中相对于 Ehdr 的偏移量
    Elf64_Addr    p_vaddr;    // Segment 起始的用户态虚拟地址
    Elf64_Addr    p_paddr;
    Elf64_Xword   p_filesz;   // Segment 在文件中占的大小
    Elf64_Xword   p_memsz;    // Segment 在内存中占的大小
    Elf64_Xword   p_align;
} Elf64_Phdr;   // 存储了程序各个 Segment 相关的 metadata
                // 你可以将 _sramdisk + e_phoff 强制转化为此类型，就会指向第一个 Phdr
```

其中相对文件偏移 p_offset 指出相应 segment 的内容从 ELF 文件的第 p_offset 字节开始，在文件中的大小为 p_filesz，它需要被分配到以 p_vaddr 为首地址的虚拟内存位置，在内存中它占用大小为 p_memsz。也就是说，这个 segment 使用的内存就是 [p_vaddr, p_vaddr + p_memsz) 这一连续区间，然后将 segment 的内容从 ELF 文件中读入到这一内存区间，并将 [p_vaddr + p_filesz, p_vaddr + p_memsz) 对应的物理区间清零。

定义函数 load_program()如下：

```
void load_program(struct task_struct *task) {
    Elf64_Ehdr *ehdr = (Elf64_Ehdr *)_sramdisk;
```

```
        Elf64_Phdr *phdrs = (Elf64_Phdr *)(_sramdisk + ehdr->e_phoff);
        for (int i = 0; i < ehdr->e_phnum; ++i) {
            Elf64_Phdr *phdr = phdrs + i;
            if (phdr->p_type == PT_LOAD) {
                uint64_t offset = (uint64_t)(phdr->p_vaddr)-PGROUNDDOWN(phdr->p_vaddr);
                uint64_t size = PGROUNDUP(phdr->p_memsz + offset) / PGSIZE;
                uint64_t ad = alloc_pages(size);
                uint64_t src_start = (uint64_t)_sramdisk + phdr->p_offset;
                for (int j = 0; j < phdr->p_memsz; ++j) {
                    *((char*)ad + j + offset) = *((char*)src_start + j);
                }
                memset((void*)(ad + phdr->p_filesz + offset), 0, phdr->p_memsz-phdr->p_filesz);
                uint64_t perm = 0x11;
                perm |= (phdr->p_flags & 1) << 3; //X
                perm |= (phdr->p_flags & 2) << 1; //W
                perm |= (phdr->p_flags & 4) >> 1; //R
                create_mapping(task->pgd, PGROUNDDOWN(phdr->p_vaddr), ad - PA2VA_OFFSET, phdr->p_memsz + offset, perm);
                // alloc space and copy content
                // do mapping
                // code...
            }
        }
        uint64_t U_stack_top = alloc_page();
        create_mapping(task->pgd, USER_END - PGSIZE, U_stack_top - PA2VA_OFFSET, PGSIZE, 23);
        task->thread.sepc = ehdr->e_entry;
        task->thread.sstatus = csr_read(sstatus);
        task->thread.sstatus &= ~(1 << 8);
        task->thread.sstatus |= (1 << 5);
        task->thread.sstatus |= (1 << 18);
        task->thread.sscratch = USER_END;
    }
```

修改 task_init()函数：

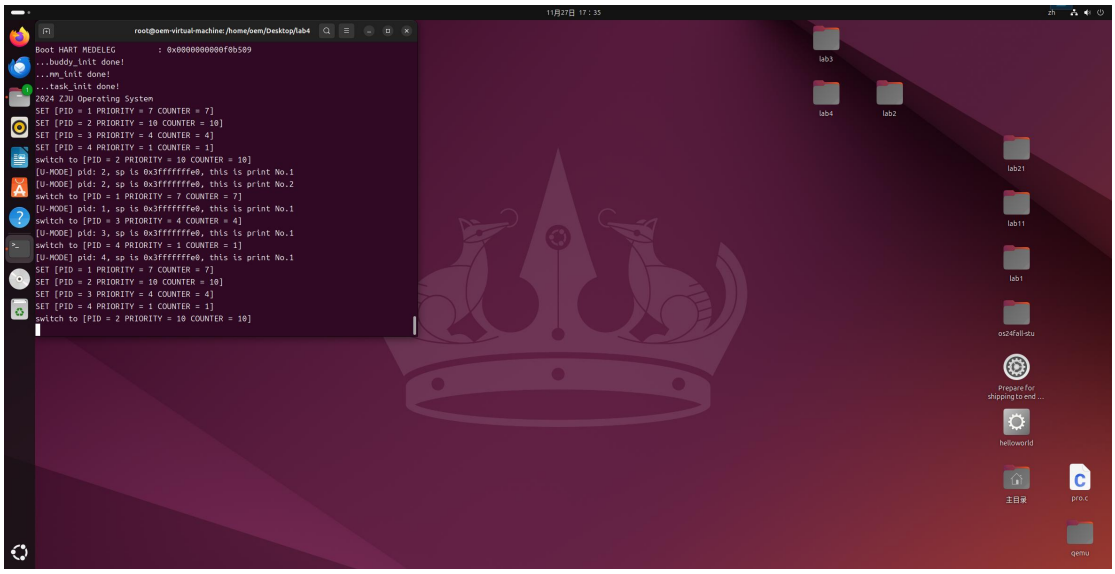
```
    void task_init() {
        srand(2024);
```

```
3.
4.          // 1. 调用 kalloc() 为 idle 分配一个物理页
5.          // 2. 设置 state 为 TASK_RUNNING;
6.          // 3. 由于 idle 不参与调度，可以将其 counter / priority 设置为 0
7.          // 4. 设置 idle 的 pid 为 0
8.          // 5. 将 current 和 task[0] 指向 idle
9.
10.         idle = (struct task_struct*) kalloc();
11.         idle->state = TASK_RUNNING;
12.         idle->counter = 0;
13.         idle->priority = 0;
14.         idle->pid = 0;
15.         current = idle;
16.         task[0] = idle;
17.
18.         // 1. 参考 idle 的设置，为 task[1] ~ task[NR_TASKS - 1] 进行初始
    化
19.         // 2. 其中每个线程的 state 为 TASK_RUNNING, 此外,
    counter 和 priority 进行如下赋值:
20.         //     - counter  = 0;
21.         //     - priority = rand() 产生的随机数（控制范围
    在 [PRIORITY_MIN, PRIORITY_MAX] 之间）
22.         // 3. 为 task[1] ~ task[NR_TASKS - 1] 设置 thread_struct 中
    的 ra 和 sp
23.         //     - ra 设置为 __dummy（见 4.2.2）的地址
24.         //     - sp 设置为该线程申请的物理页的高地址
25.
26.         for (int i = 1; i < NR_TASKS; i++){
27.             task[i] = (struct task_struct*) kalloc();
28.             task[i]->state = TASK_RUNNING;
29.             task[i]->counter = 0;
30.             task[i]->priority = rand() % (PRIORITY_MAX - PRIORITY_MIN
     + 1) + PRIORITY_MIN;
31.             task[i]->pid = i;
32.             task[i]->thread.ra = (uint64_t) &__dummy;
33.             task[i]->thread.sp = (uint64_t) task[i] + PGSIZE;
34.
35.             task[i]->pgd = (uint64_t)alloc_page();
36.             for (int j = 0; j < 512; j++){
37.                 task[i]->pgd[j] = swapper_pg_dir[j];
38.             }
39.
40.             load_program(task[i]);
41.         }
```

```
42.
43.         printk("...task_init done!\n");
44.     }
```

（九）编译及测试

测试结果如下：



根据图中显示，测试结果正常。

三、讨论和心得

本次实验建立在 lab3 环境已经搭建好的基础上，逻辑上稍微有些复杂，需要查阅 risc-v 手册了解 sstatus 寄存器每个位代表什么以及正确地处理中断。本实验有两个中断，一个是时钟中断，另一个是用户态程序调用系统调用函数中断，需要设置正确的执行函数。

这次实验让我对虚拟内存的工作机制有了更深入的了解，同时也加深了对用户态进程的理解。

四、思考题

1.
一对一。每一个用户态进程对应的内核态的栈是 task[i]->thread.sp = (uint64_t) task[i] + PGSIZE，这说明不同用户态线程所转到的内核态线程是不同的。

2.
trap_handler 执行前后会进行寄存器的保存和恢复，使得被中断的程序能回到原来的状态。但是系统调用需要改变寄存器，如果不通过 reg 结构体改变栈上保存的内容，修改的数据就被后续恢复的过程覆盖了。

3.
异常发生时，sepc 会保存触发异常的指令地址，当用户态指令触发 ecall 时，sepc 指向的将是 ecall 指令的地址，ecall 指令在执行时会停止当前程序的执行，从用户态进入内核态，因此，处理完系统调用后，sepc 需要加 4，使得程序从系统调用返回时继续执行 ecall 之后的下一条指令，如果不加 4 会再回到同一位置触发 ecall 导致死循环。

4.
p_filesz 指的是 Segment 在文件中占的大小，p_memsz 指的是 Segment 在内存中占的大小。程序段在内存中的映射区域包含比文件中更多的空间，内存中除了程序可能还包含未初始化或初始化为 0 的数据，这些可能是因为内存分配时需要分配若干个整页导致的，这就造成了 p_memsz 可能大于 p_filesz。

5.
只要不同进程的虚拟地址映射页表内容不同，即使它们使用相同的虚拟地址，指向的物理地址也不同。
没有常规的方法，只有在内核态才能知道。

五、附录

无